13

Computers can learn

Neural networks

The statement we made in Chapter 1 about computers being models of the human brain is obviously a great simplification. The human brain is so complex and functions at so many levels that mankind is still very far from fully understanding its workings. Therefore, we are only able to model some aspects of its functionality and in a very limited way.

The architecture proposed by von Neumann, the basis of all modern computers, is one such simplified model. Its two main components—the central processing unit (CPU) and the memory—are analogous to the two main functions of our brain—thinking and remembering. A very simplified model of our thinking process is realized by the CPU and it is a cyclic repetition of the following actions.

- Read an instruction from memory.
- Read the data required to carry out the instruction from memory.
- Carry out the instruction.
- Write the resulting data to memory.

A little-known fact is that the first digital programmable computer (the Z3) was produced in 1941, before and independently of John von Neumann, by Konrad Zuse. Zuse built his first computer (the Z1) in his parents' living room in 1938. The Z3 had all of the basic features of modern computers; it was programmable and used floating-point arithmetic. It was not until much later, however, that Zuse's ground-breaking achievements were discovered and acknowledged.

Konrad Zuse (1910–1995); German engineer and a computer pioneer ahead of his time. Designed and built the first computer and developed the first high-level programming language–Plankalkül.

We need not convince the reader of how effective this model prove to be—the range of applications of modern computers speaks for itself. Yet it is pretty clear that computer 'brains' are still a long way from ours. A major shortcoming of the von Neumann architecture is its sensitivity to errors—even a small change to the set of instructions for the CPU (the computer program) usually causes the program to stop functioning altogether. Another limitation is the sequential nature of the computing process—the fact that the processor can only do one thing at a time. Yet another deficiency is the computer's inability to adapt to changing circumstances. All of these issues are addressed by an alternative computer model—the **neural network.**

Neural networks, besides trying to model the functionality of the brain, also try to imitate its structure. A brain is a network of many interconnected components which function in parallel and communicate with each other. In other words, it is a **distributed system**. Distributed systems are usually more resistant to errors and damage than centralized ones. A centralized system always has a weak point—its center—the damage of which renders the whole system useless. Distributed systems have no single weak point and can often continue functioning after a part of them has been corrupted or destroyed.

In fact, this motivation was the reason why the US Defense Department created the experimental computer network called ARPAnet (named after the Advanced Research Projects Agency which funded the project). The network's distributed architecture was supposed to assure its functionality, even in the case of extensive damage (brought on, for example, by a Soviet nuclear attack). What nobody expected was that the ARPAnet was going to evolve into a giant worldwide computer network called the Internet, and that the main threats to its smooth functioning were going to be the authors of computer viruses and spam mail rather than a foreign enemy. Still, the distributed nature of the network fulfilled its expectations; the Internet does not cease to function even when a virus immobilizes a large number of its nodes.

Imitating the brain

A human brain (as well as an animal one) consists of specialized cells called neurons. Neurons are capable of passing information to and from each other.

Besides all of the typical cell elements, they are equipped with dendrites and an axon (see Fig. 13.1). Dendrites collect signals from the surrounding neurons. When the total intensity of the gathered signals passes a certain threshold, an electrical impulse (called the action potential) is propagated down the axon. Once the signal reaches the end of the axon (called the axon terminal), it can be picked up by neurons located in the vicinity of the terminal. Some neuron axons are short, but some can be very long.

Signals do not pass directly from one neuron to another. Each neuron connection is actually a tiny valve called a synapse. Some synapses pass the actual electrical signals between neurons, but most pass chemical substances called neurotransmitters. The action potential triggers the axon terminal to release neurotransmitters into the synapse. These disperse and bind to the receptors of the receiving neurons, effectively transmitting information.

When a signal traverses a synapse it may become stronger or weaker, depending on the properties of the synapse. These synapse strengths influence the way in which the brain processes information. On the other hand, processes going on in the brain can cause the synapse strengths to change. These two facts lead to the hypothesis that the adjustment of synapse strengths is the basis of memory and learning. This is called the Hebbian theory after the

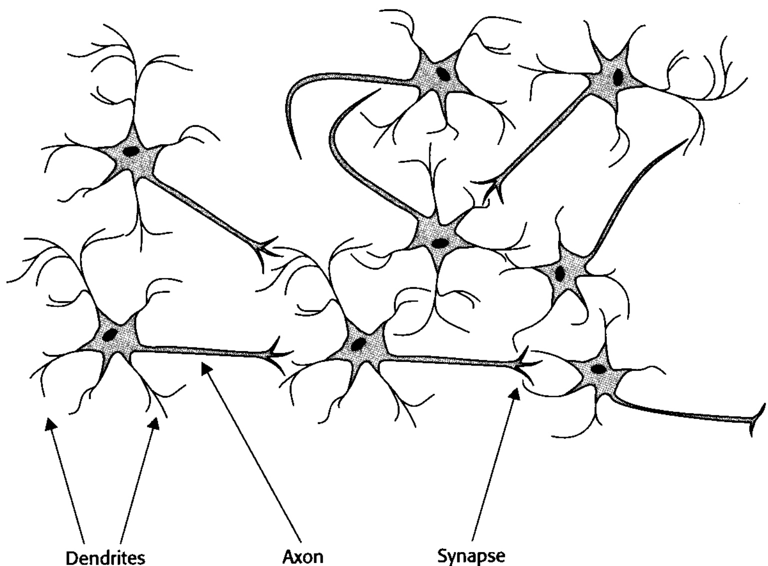


Fig. 13.1 A schematic diagram of neuron connections in the brain. In reality each neuron is connected to thousands of others, forming a much more complicated three-dimensional mesh.

Canadian psychologist Donald Hebb who formulated it in 1949, thus founding what is now called cognitive psychology.

A single neuron is very slow compared to a modern computer processor. It takes about one millisecond (10^{-3} s) to react to an impulse, while a 1 GHz processor performs one instruction in a nanosecond (10^{-9} s). However, if we take into account that a human brain is composed of roughly 10^{11} simultaneously functioning neurons and that each neuron is connected to thousands (sometimes even hundreds of thousands!) of other neurons, then we can safely say that a brain has many orders of magnitude more computing power than any computer created so far.

The former description of the workings of our brain is highly simplified. In fact, it is yet another model which mirrors just a few aspects of the process, leaving many elements out. This simple model, however, was what inspired scientists to build **artificial neural networks**, called neural networks or neural nets for short. Artificial neural networks are composed of artificial interconnected neurons. Each artificial neuron connection, the parallel of a synapse, has a numeric weight associated with it; as a signal (which is also a number) passes through the connection it is multiplied by this weight. An artificial neuron receives incoming signals, applies some simple function to their sum, and outputs the result.

The function used by artificial neurons to transform the sum of inputs into an output is called the **activation function**. As we mentioned before, real neurons fire when the sum of their inputs reaches a certain threshold. Imitating this behavior means applying a **threshold** activation function (see Fig. 13.2) of the general form

$$f(x) = a \text{ sign } (x) + c.$$

This function is often used as an activation function in artificial networks. Its main advantage is simplicity, but it does have several drawbacks, such as

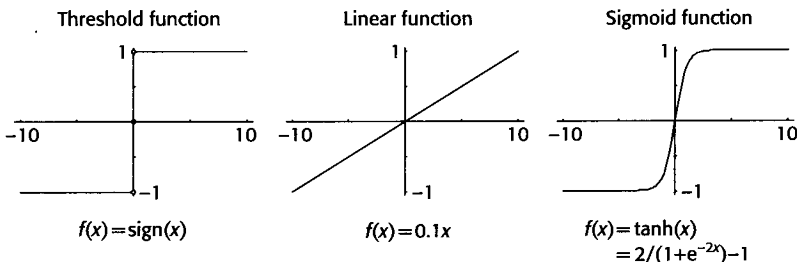


Fig. 13.2 Three kinds of activation functions used in neural networks.

a discontinuity at the threshold point. An alternative is the **linear** function (see Fig. 13.2)

$$f(x) = ax + c.$$

The most widely-used activation functions, however, are the **sigmoid** functions (see Fig. 13.2), a 'compromise' between the former two:

$$f(x) = \frac{a}{1 + e^{-bx}} + c.$$

The most common sigmoid functions are the simplest, i.e. $f(x) = 1/(1 + e^{-x})$ or the hyperbolic tangent $f(x) = 2/(1 + e^{-2x}) - 1 = \tanh(x)$.

Often, an additional value is associated with each neuron and is called a **bias**. The bias is added to the sum of the neuron's inputs before applying the activation function, effectively shifting the plot of the activation function left or right and shifting the threshold if there is one. For unifying the terminology, instead of associating extra values with neurons, additional neurons are introduced into the network. These additional neurons are called bias units; they have no inputs and always output 1. Each neuron is at the output end of a connection with a bias unit and the weight of that connection is the bias of the neuron. The advantage of this approach is that it makes it easy to adapt an existing learning mechanism (which will be described later) to alter the bias values as well.

Neurons in the brain are often classified as sensory neurons, motor neurons, and interneurons. Interneurons just pass information from neurons to neurons, as described earlier. Sensory neurons receive information from external stimuli rather than from other neurons. Motor neurons, in turn, pass information on to muscles, causing them to contract. Sensory and motor neurons permit a two-way interaction between the brain and the environment. Likewise, an artificial neural network has a set of input nodes, a set of output nodes, and a set of internal nodes (usually called hidden nodes). While the simplest neural nets do not contain any hidden nodes, all must have some input and some output nodes to serve their purpose.

Several companies produce physical neural networks from arrays of simple processors. Due to their lack of flexibility, however, the market for hardware neural networks is still very small. A much more popular approach is emulating neural networks with a traditional von Neumann computer. Software neural networks are computer programs which simulate the behavior of an actual network. Such programs can no longer take advantage of the parallel design of neural networks, since most of them end up running on single-processor machines. Yet they are much easier to manage than hardware

networks and are widely used when the ability to adapt and fault tolerance—rather than parallelism—are the reason for choosing a neural net solution.

There are two main difficulties when trying to solve a problem using a neural network.

- **Designing the network**—deciding on the number of neurons and specifying the connections between them; in other words, constructing a directed graph representing the network.
- **Teaching the network**—adjusting the weights on the basis of some sample input so that the network will 'learn' to serve its purpose.

We are still pretty much in the dark about how nature 'solves' these problems. Apparently, the 'design' of our brains is largely the product of evolution, though the connections between neurons do change throughout our lives. There are a few theories on how different kinds of stimuli can affect synapse strengths resulting in learning, but nobody knows how the process works exactly. We cannot even be sure whether synapse strengths are indeed the basis of our memory.

Neither is there a unique methodology for designing and teaching artificial neural networks. In general, a neural network design can be any directed graph. This large degree of freedom makes it very difficult to formulate specific theories on the subject. Most often, additional restrictions are introduced and theories are developed for certain narrowed-down families of networks. In the remainder of this chapter we will describe a few of the most popular neural network families and the algorithms associated with them.

Perceptrons

The simplest and most common type of neural network is a **multi-layer perceptron** (MLP). MLPs are **feedforward** networks, meaning that they do not contain any cycles. Information supplied to the input nodes passes through a feedforward network once, until it reaches the output nodes, and then the state of the network stabilizes. An MLP is organized into a series of layers: the input layer, a number of hidden layers, and an output layer. Each node in a layer is connected to each node in the neighboring layers. These connections always point away from the input and in the direction of the output layer. See Fig. 13.3 for the topology of a sample two-layer perceptron.

Some people would call the network in Fig. 13.3 a three-layer perceptron, since it has three layers of nodes. Most often, though, one refers to the number of connection layers, rather than node layers, when specifying the layer count of a perceptron. The term perceptron used alone often refers to a single-layer perceptron (SLP), with one input layer, one output layer, and no hidden layers.

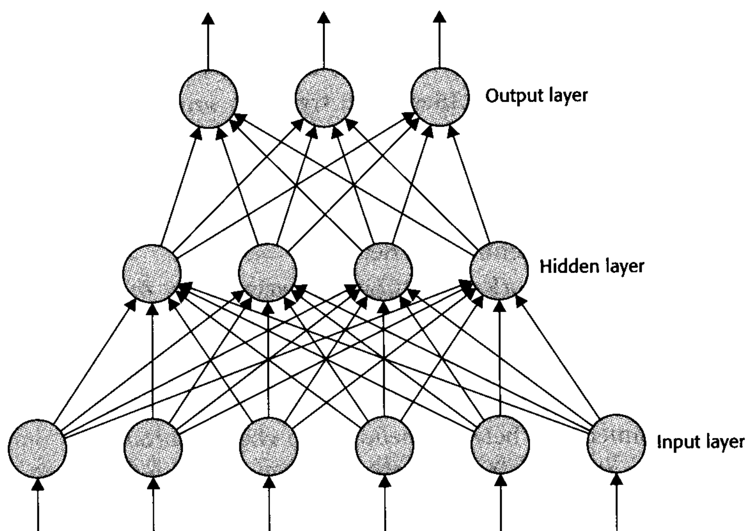


Fig. 13.3 A two-layer perceptron.

An MLP transforms signals according to the algorithm in Fig. 13.4. To better understand this procedure and the associated notation, let us consider the simple two-layer perceptron depicted in Fig. 13.5. This network, given two binary values (0 or 1), outputs the *exclusive or* of the two values. Exclusive or, XOR for short, is a logical function defined as follows:

$$\mathrm{XOR}(x, y) = \begin{cases} 0, & \text{when}\, x = y, \\ 1, & \text{when}\, x \neq y. \end{cases}$$

In their 1969 work entitled 'Perceptrons', two pioneers of artificial intelligence, Marvin Minsky and Seymour Papert, proved that no perceptron without hidden layers can calculate the XOR function. They conjectured that the same holds for more complicated perceptrons as well, which for a time significantly cooled the interest in neural networks altogether. Fortunately, as the above example confirms, MLPs are not quite as impotent as Minsky and Papert believed.

Marvin Minsky (1927–); a founder of artificial intelligence; among his many inventions are the confocal scanning microscope and the Logo turtle (together with Papert).

- For each node N_i^1 in the input layer (layer number 1), set its signal S_i^1 to I_i, where I_i is the ith component of the input signal.
- For each layer k, starting from the layer after the input layer (layer number 2) and ending with the output layer (layer number M):
  –For each node N_i^k in layer k, sum the incoming signals multiplied by the corresponding connection weights, apply the activation function f, and set the signal of the node S_i^k to the result. In other words, for each node N_i^k in layer k, set its signal to

$$S_i^k = f\left(\sum_{j=1}^{L_{k-1}} w_{j,i}^{k-1} S_j^{k-1}\right),$$

  where L_{k-1} is the number of nodes in layer $k-1$ and $w_{j,i}^{k-1}$ is the weight of the connection from node N_j^{k-1} to node N_i^k.
- For each node N_i^M in the output layer (layer number M), copy its signal S_i^M to O_i, where O_i is the ith component of the output signal.

Fig. 13.4 The algorithm used by an $(M-1)$-layer perceptron to transform the L_1-long input vector $\{I_1, I_2, \ldots, I_{L_1}\}$ into the L_M-long output vector $\{O_1, O_2, \ldots, O_{L_M}\}$.

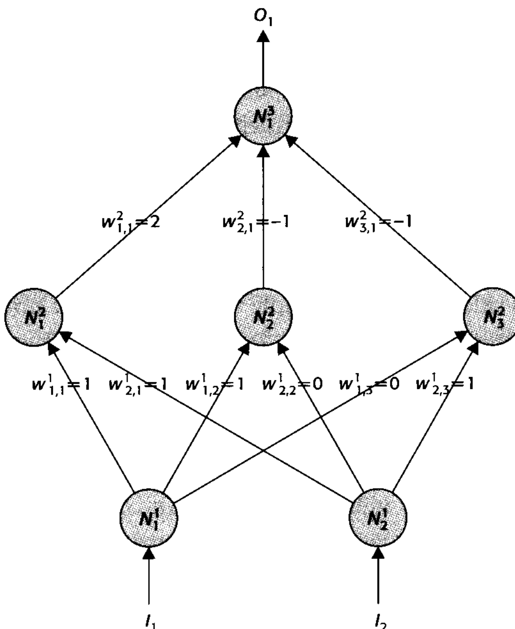


Fig. 13.5 A simple MLP which calculates the XOR of two binary values. Its activation function is $f(x) = \text{sign}(x)$.

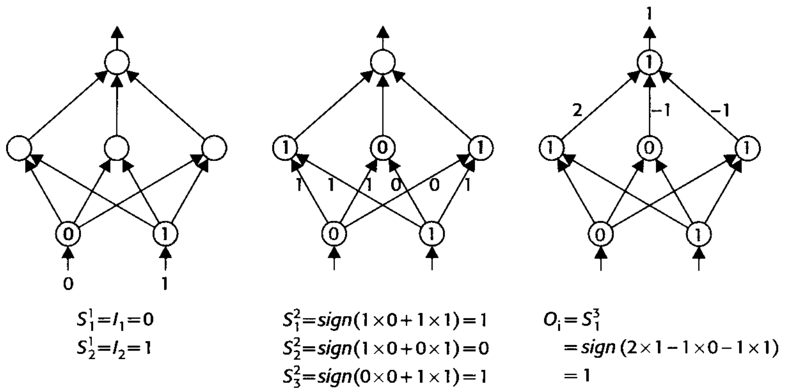


Fig. 13.6 A calculation performed by the MLP in Fig. 13.5. The network calculates that XOR(0, 1) = 1.

Seymour Papert (1928–); American mathematician; inventor of the Logo programming language.

Figure 13.6 shows how the subsequent steps of the algorithm from Fig. 13.4 applied to the network depicted in Fig. 13.5 lead to the result XOR(0, 1) = 1. As an exercise, we encourage the reader to verify that the network also produces the correct results when the pairs (0, 0), (1, 0), and (1, 1) are taken as input.

The network in Fig. 13.5 provides one of the most complicated ways to calculate the XOR of two binary values using a computer. After all, the same can be achieved with just one elementary processor instruction. The aim of the example was to familiarize the reader with neural networks and the related terminology, but it did nothing to demonstrate why neural networks are actually useful. We contrived both the network topology and the connection weights to achieve the desired result. It was not a simple task and what we got in the end was a very complex way to calculate a very simple function. We did not take advantage of the main feature of neural networks, namely their ability to *learn.*

Teaching the net

Teaching neural networks is the process of adjusting the network's connection weights in order to make it function in the desired way. In the former example, we picked the weights *by hand* to make the network function in the desired

way (calculate the XOR of two binary numbers). This approach may be good in some cases of hardware neural networks, but it is usually very inefficient. It is much easier to design an algorithm rather than to invent a network for solving a problem. When referring to the process of teaching a neural network, one usually means applying some **learning algorithm** which adjusts the network's weight *automatically*. There are different learning algorithms suitable for different tasks and for different network topologies.

Perhaps the simplest learning algorithm comes from John Hopfield and is based on Hebb's observations of actual processes going on in the brain. Hebb noticed than when two neurons are connected by a synapse and when these two neurons tend to fire at the same time, then the strength of the synapse increases. Hebb's theory is that this process lies at the base of **associative memory**—the memory which associates remembered objects with each other (like one may associate the smell of lilacs with spring).

John J. Hopfield (1933–); a physicist at California Institute of Technology and Princeton University. In 1982 he presented a new formulation of neural network theory based on statistical physics.

The simplest Hopfield network acts as an *autoassociative* memory, which means that it associates objects with themselves. Given an input pattern, it produces the same pattern as output. Producing the same output as input may seem useless; but notice that neural networks tend to be fault tolerant. So when a slightly *corrupted* version of a memorized pattern is inputted, the network will still output the original memorized pattern. For example, one can teach the network with a set of letters encoded as bitmap images. Later, when scanning a text, one can run each scanned letter through the network to remove the errors which the scanning process introduced. One could achieve the same by comparing each scanned letter with the letters from the set and replacing it with the letter from the set most similar to it. The neural network solution, however, can be implemented in hardware and is then much faster than the direct comparisons.

A simple Hopfield network for recognizing binary patterns (stored as sequences of -1s and $+1$s) of length n is a single-layer perceptron with n input nodes and n output nodes. The activation function is $f(x) = \text{sign}(x)$.

An alternative topology of Hopfield nets identifies its input nodes with its output nodes, so that the network has just n input/output nodes and every node is connected to every other. The full graph model is more powerful since it allows the

network to be treated as a recurrent one. Recurrent networks will be described later in this chapter.

We will use the term **training set** to mean the sample data used during the learning process. In this case, the training set is a set of patterns to memorize $\{P^1, P^2, \ldots, P^m\}$, where the kth pattern is a sequence $P^k = \{p_1^k, p_2^k, \ldots, p_n^k\}$ of $+1$ and -1 values. Training the network on this set is achieved by setting the following weights:

$$w_{i,j} = \frac{1}{n} \sum_{k=1}^{m} p_i^k p_j^k.$$

This simple rule can be interpreted as follows. For each pattern in the set (P^k) and for each connection (between input node i and output node j), *add* to the connection weight the term $1/n$ if there is a positive correlation between the values at the positions i and j in the pattern ($p_i^k = +1$ and $p_j^k = +1$, or $p_i^k = -1$ and $p_j^k = -1$) and *subtract* from the connection weight the same term if there is a *negative* correlation between the values at the positions i and j in the pattern ($p_i^k = +1$ and $p_j^k = -1$, or $p_i^k = -1$ and $p_j^k = +1$). The correlation information is stored in the network weights so that, when later presented with a somewhat perturbed input, the network is able to reproduce the original pattern. Of course, in order for this method to work properly, the patterns in the training set must by distinctly different from each other and the perturbation of the input cannot be too great.

Hopfield's learning method is simple—it provides a ready formula for all of the weights—but it is also limited in its applications. It can only make the network output identical to its input, perhaps filtering some noise on the way. What if we want the network to compute something else?

Fifteen years after the invention of neural networks, the first algorithm for training them was found by the American psychologist Frank Rosenblatt. Rosenblatt described a perceptron in its simplest form (an input layer and an output layer with the sign activation function) and simulated it in 1958 on an IBM 704 machine. He also described a training algorithm for it, namely the **perceptron learning rule** (PLR).

PLR uses a training set consisting of pairs of *input* and *desired output*. At first, all of the weights in the network are set to random values. Next, a series of iterations is performed. In each iteration an input pattern $\{I_1, I_2, \ldots, I_n\}$ from the training set is fed to the network. The network produces some output $\{O_1, O_2, \ldots, O_m\}$ from this input, and that output is subtracted from the desired output $\{D_1, D_2, \ldots, D_m\}$ to obtain the error vector $\{E_1, E_2, \ldots, E_m\}$. The length of the error vector is a measure of how well the network behaved. If

it is zero then the network produced exactly the desired output. The longer the error vector, the worse the behavior of the net. Every network weight is subsequently adjusted, depending on the component of the error for which it was responsible. That is, if $E_j = 0$ then $w_{i,j}$ stays unchanged, but if $E_j = \pm 1$ then $w_{i,j}$ is modified. The increment or decrement of the weight is controlled by a small parameter called the **learning rate** (customarily denoted by η) and it is additionally proportional to the input of the connection I_i. Thus, the modified weight is set to be $w_{i,j} + \eta I_i E_j$. The learning rate should be some small positive number, such as 0.1. If the learning rate is too high, then each iteration will cause the network to 'forget' what it has learned during previous iterations. If the learning rate is too low, then the learning process will take a very long time. The PLR algorithm is detailed in Fig. 13.7.

Iterating the algorithm over and over tends to decrease the total error of the network...up to a point. Ideally, that point is when the total error is zero, and so the network produces the exact desired output for each pattern in the training set. The point is that such a trained network should also be able to behave in the desired way when provided with patterns which were not in the training set. Whether it actually does that depends on the nature of the problem which we are trying to solve. Simple perceptrons are good enough for many applications. They are capable of learning only simple functions; but, on the other hand, they work for functions with any number of arguments.

Let us consider the problem of optical character recognition (OCR). We are seeking a function which will transform a bitmap image into a single letter. This function is not complex from the mathematical point of view. Still, it is a function of very many arguments—as many as there are pixels in the bitmap. It would be very hard for a human to figure out how each particular pixel influences what the resulting letter is and therefore to program a computer to recognize letters.

The PLR provides a mechanism for teaching the computer to recognize letters by presenting it with a set of example bitmap–letter pairs. At no point do we have to actually know the *mechanism* of transforming bitmaps into letters—the network can 'figure it out' on its own, with a certain degree of error, of course.

Bernard Widrow and Marcian E. Hoff described a more general rule for teaching perceptrons, namely the **delta rule**, known also as the **Widrow–Hoff rule**. It is based on the idea of **gradient descent**.

Gradient descent is a way of minimizing an error when this error depends on many factors. This is exactly the case in neural networks; the network produces some error when acting on inputs from the training set and this error value depends on

1. Set all weights to random values.
2. Adjust the weights.
   a) Set $E_{\text{total}} = 0$.
   b) Adjust the weights according to one pattern from the training set.
      (i) Take the next input vector $\{I_1, I_2, \ldots, I_n\}$ and its corresponding desired output vector $\{D_1, D_2, \ldots, D_m\}$ from the training set.
      (ii) Apply the feedforward algorithm in Fig. 13.4 to obtain the perceptron's output $\{O_1, O_2, \ldots, O_m\}$ from the sample input $\{I_1, I_2, \ldots, I_n\}$.
      (iii) Subtract the actual output from the desired output to obtain the error vector, i.e. $E_i = D_i - O_i$ for $i = 1, 2, \ldots, m$.
      (iv) Add the squared length of the error vector to the total error:

$$E_{\text{total}} = E_{\text{total}} + (E_1^2 + E_2^2 + \ldots + E_2^0).$$

      (v) For each weight $w_{i,j}$ connecting the ith node of the input layer with the jth node of the output layer, correct the weight by setting

$$w_{i,j} = w_{i,j} + \eta I_i E_j.$$

   c) If there are still patterns in the training set then return to step 2(b).
3. If the total error E_{total} is lower than the previous time that this point was executed, then go back to step 2.

Fig. 13.7 The perceptron learning rule. The parameter η is the learning rate.

all of the network's weights. What we want to do is to find the lowest value of the error function. What makes this problem technically difficult is the large number of variables–all of the network's weights. The gradient descent rule tells us to start from any random point (this is equivalent to initializing the network with random weights) and move along each axis proportionally to the negative slope of the plot along that axis. To visualize this process, imagine standing on the side of a hill. Going north or west leads uphill, and going south and east both lead downhill, but east is twice as steep as south. In this case gradient descent would tell us to make one step south and two steps east–if we want to get to the bottom, that is.

Applying gradient descent to correct the behavior of a single-layer perceptron means calculating, for each weight, the partial derivative of the error with respect to that weight and then correcting the weight proportionally to the result. The partial derivative tells us the direction and the steepness of the slope of the error plot along the axis corresponding to that weight. In other words, it tells us in which direction and how strongly this particular weight influences the error. The formula for correcting the weights of a single-layer perceptron with activation function $f(x)$ using gradient descent is

$$w_{i,j} = w_{i,j} + \eta f'\left(\sum_{k=1}^{n} w_{k,j} I_k\right) I_i (D_j - O_j),$$

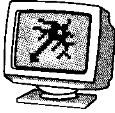
where η is the learning rate. It differs from the perceptron learning rule by the derivative of the activation function $f'(\sum w_{k,j} I_k)$. This term is 'almost' the output of the jth output node; but, instead of the activation function, the derivative of the activation function is evaluated at the sum of the weighted inputs. The gradient descent method requires the activation function to be differentiable, which is not the case for the discontinuous $f(x) = \text{sign}(x)$ function. For linear activation functions the derivative gives just a constant and we obtain the same results as before.

As Minsky and Papert showed, single-layer perceptrons are by their nature limited in what they can do. They cannot even calculate the simple XOR of two Boolean numbers! Multi-layer perceptrons are more powerful. The question remains as to how to train a multi-layer network.

The most commonly used learning algorithm for MLPs, and probably for neural networks in general, is **backpropagation**. Backpropagation is similar to the perceptron learning rule in the sense that it starts from random weights and uses a set of *input* and *desired output* pairs to gradually correct these weights. The reason that it is called backpropagation is that the weights leading to the output layer are corrected first, then the weights before them, and so on, until the layer at the bottom is reached. The order of correcting weights is backwards with respect to the order in which the signals are calculated when the network performs its task.

Just like the delta rule, backpropagation is based on applying gradient descent (see the previous note) to the network's error. The corrections to the weights closest to the output can be calculated just as in the case of the single-layer perceptron. The corrections for the weights positioned deeper inside the network are harder to calculate because each deeper weight influences all of the output nodes rather than just one of them. Fortunately, it turns out that we can use the partial results

from calculating the corrections in one layer to significantly simplify the calculations needed for the previous layer—hence the backwards direction of the algorithm.



Using the program **Hopfield** you can design and train a neural network to recognize your handwriting. You can build a training set consisting of sample letters, choose the design of the network, and train the network on the prepared training set. A trained network is capable of recognizing your handwritten letters even though no two letters you write will ever be exactly the same. The network is a multi-layer perceptron with a sigmoid activation function and bias weights. The training algorithm used is backpropagation with momentum. See the *Modeling reality* help file for more information on the features and implementation of this program.

Recurrent networks

As we mentioned earlier, perceptrons are feedforward networks, meaning that the signal passes through such a network once—from the input to the output nodes. **Recurrent** networks contain cycles, so that information can pass back and forth between nodes. This feature makes recurrent networks potentially much more powerful, but also makes it harder to find algorithms for such structures.

The simplest recurrent network is an Elman network, named after its inventor Jeff Elman. An Elman network is a two-layer perceptron with additional nodes called context units. There are as many context units as there are hidden nodes. They are located on the level of the input nodes and connect with all of the nodes in the hidden layer as if they were additional input nodes. Additional **feedback** connections (of weight one) lead from the nodes in the hidden layer to the context units (see Fig. 13.8). Each time the network processes some input, the state of the hidden layer is 'stored' in the context units. This state is fed together with the input the next time the network processes something.

While feedforward networks always produce the same output given the same input, the output of recurrent networks will depend on the current input as well as on the previous inputs. This feature is convenient when operating on data which naturally comes in a series—such as stock market data, for example. Suppose that a neural network is designed to predict the closing value of a company's stock on a particular day. It takes as input the opening value of the stock, the values of other related stocks, and, say, the exchange rate of the dollar. If the network is a standard feedforward network, then it will base its estimate on just that input. If it is an Elman network, then it will base

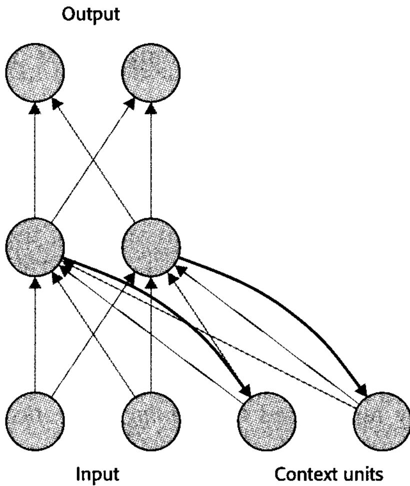


Fig. 13.8 A simple Elman network with feedback connections marked by thicker lines.

its estimate on that input and on the values from the previous days. The way in which it takes the previous results into account is also subject to training. The Elman network in its simplest version can be trained using the standard backpropagation algorithm, since the feedback connection weights are fixed and do not need to be trained.

As we mentioned earlier, the simple Hopfield nets come in a more powerful recurrent form. The learning algorithm for recurrent Hopfield nets remains the simple one, where we set all of the weights at once using a simple formula. When processing, however, rather then running the input once through the network, we run it through continuously, providing the output as input again, and so on. This process will eventually converge to the point at which the output coincides with one of the memorized patterns, and so running it through the network does not change it.

Different types of training

There are three types of learning algorithms for neural networks: **supervised, reinforcement**, and **unsupervised**.

All of the learning algorithms which we saw earlier in this chapter are examples of supervised learning. Supervised learning is based on presenting the network with examples of input and the desired output. In the case of Hopfield networks the desired output was assumed to be equal to the input.

During reinforcement learning, the network is still presented with example input, but, instead of the correct answer being exposed, the network is just

given a 'grade' which reflects how well it performed when processing that input. This kind of training was invented to mimic the learning process of humans.

Applying an evolutionary algorithm, such as the ones described in Chapter 12, to find the weight values for a network is an example of reinforcement training. A straightforward genetic algorithm for training a network is based on coding all of the weights in the genotype and then applying the standard evolutionary techniques to find the most suitable weight set. This approach requires the ability to compare a network (identified with a weight set) with other networks in order to give it a reproductive advantage, which is equivalent to grading the network based on its output.

Evolutionary algorithms are relatively rarely used for *training* networks, due to the availability of alternative algorithms such as the gradient descent method. Their much more important application is in *designing* neural networks. We mentioned several algorithms for finding network weights, but bypassed the problem of choosing the network topology (for example, the number of hidden layers in a perceptron). The truth is that there are no methods, strictly speaking. Despite the existence of some empirical truths on the subject (for example, that one hidden layer is enough for most things), every rule has its exceptions. Evolutionary algorithms prove to be a very good tool for optimizing the topology of a neural network.

In unsupervised training the network is presented with some sample input and no additional information is provided. One might wonder how unsupervised learning can teach the network to behave in the way that we want, since we do not specify what we want at any time. This is just the point; sometimes we do not even know ourselves what we want to achieve when analyzing a set of data. A neural network can then help us search for some sort of patterns and find a classification system for the data. Classifying data like this is called **clustering**. Clustering is the process of dividing a set of objects into groups of similar ones. In 1981 Teuvo Kohonen described a neural network which, through unsupervised learning, finds the structure underlying a set of data. When later presented with an input, it can show where this input lies within the structure. This type of network is called a **self-organizing map** (SOM) or a **Kohonen map**. SOMs are called maps because they visualize high-dimensional data in a low-dimensional space—most often on a plane. High-dimensional data describes objects which have many attributes. Such objects could be sound patterns, visual patterns, text strings, or anything else. An SOM learns to position each such object on its low-dimensional grid in such a way that similar objects are close to each other and ones which differ more are farther away.

Summary

The idea of neural networks is even older than that of the von Neumann computer, and has seen many ups and downs over the years. In 1943 Warren McCulloch and Walter Pitts described the artificial neuron and hypothesized about networks of such neurons, but could not do much with the technology available at that time. In the late 1950s Frank Rosenblatt simulated a trainable perceptron on an early computer, thus stimulating a great interest in neural networks. This interest waned at the end of the 1960s with Minsky's and Papert's findings concerning perceptron limitations, only to be reborn in the 1980s with the advent of the PC and new developments in neural network theory (such as Hopfield's work). In recent years neural networks have found a wide range of applications in economy, finance, medicine, science, sports, and other many other fields of human activity. Many companies worldwide specialize in producing neural network software for different purposes. The following are several examples of specific neural network applications, which we found by browsing the websites of such companies:

- currency price prediction,
- stock market prediction,
- investment risk estimation,
- direct marketing response prediction,
- identifying policemen with a potential for misconduct,
- jury summoning,
- forecasting highway maintenance,
- medical diagnosis,
- horse race outcome prediction,
- solar flare prediction,
- weather forecasting,
- product quality testing,
- speech synthesis,
- speech recognition,
- character recognition,
- signature verification,
- driving automation,
- chicken feed compound selection,
- finding gold,
- pollution monitoring,

...and these do not exhaust the list. Some even use neural networks to predict lottery results. While this particular application is not (understandably)

effective, most of the others can give very good results. The theory behind neural networks is still being developed by scientists. Meanwhile, the ability to apply this theory in practice (to choose a network design, training algorithm, and other parameters) has developed into a very valuable skill, whose main component is a special type of intuition rather than sound knowledge.